SAM & MAX

Facebook Social Plugins
Gravatar
Piwik Analytics
Reddit
Tumblr Buttons
Twitter Button
Wordpress Stats

Search this site...

Programmation | Admin System | Philo et culture | Cul | Cours et tutos | Nos tweets SANS twitter :-) | Mu | Nos githuberies

Home » Programmation » Comment fonctionne HTTP ?

Comment fonctionne HTTP ?

Posted by Sam on 18 Jun 2014 in Programmation | 6 comments

Plus je fais du dev Web, plus je m'aperçois que beaucoup de mes collègues n'ont aucune idée de comment fonctionne HTTP sous le capot. Comme vous le savez, ma règle numéro 1 c'est qu'**il n'y a rien d'évident**, donc petit tuto pour expliquer les bases.

On ne va pas rentrer dans les petits détails, juste une petite intro histoire de savoir ce qui se passe derrière ce script PHP ou cette application bottle.

Article long, vous connaissez la chanson.

Et puis c'est dans le ton de l'actu ^^

La logique de client / serveur

(**Je me repompe moi-même**)

Se balader sur le Web, c'est comme aller au resto. On est un client, on demande quelque chose au serveur, le serveur va voir en cuisine, et revient avec la bouffe, ou une explication sur l'absence de la bouffe (mais jamais pourquoi la bouffe est dégueulasse, allez comprendre):

Une lecture plus fastoche ?

Télécharger cette page en PDF

Follow us @sam_et_max

Souscrire à nos conneries

Entrez votre adresse mail et vous recevrez une notification à chaque nouvel article.

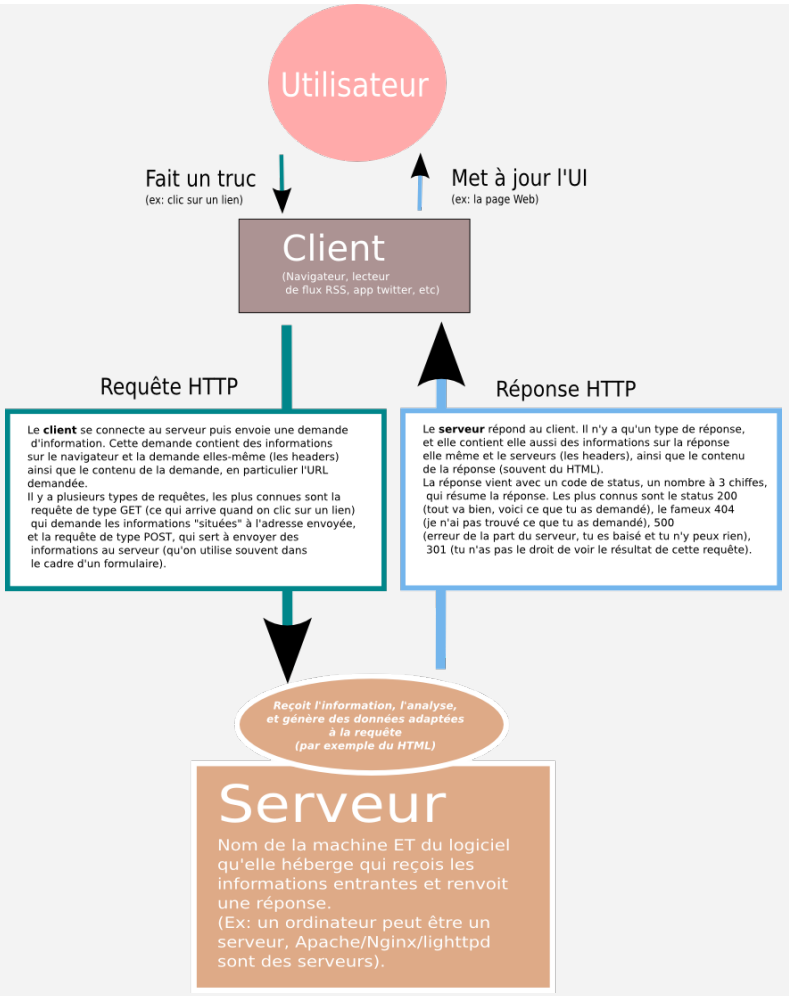
Rejoignez les 324 abonnés

françis@ça-mousse.com

J'adhère bon dieu!

Posts Populaires

- On a un nouveau projet et on a besoin d'UNE info sur vous (265)
- Entreprendre en France, Autopsie d'un échec. (114)
- Un gros Troll de plus sur Javascript (108)
- FizzBuzz en Python (98)



Le protocole HTTP, en (très) gros

Ce cycle requête/réponse se déroule des centaines de fois quand on parcours un site Web. Chaque lien cliqué déclenche une requête GET, et la page suivante ne s'affiche que parce qu'on a reçu la réponse contenant le HTML de celle-ci. Les formulaires, les requêtes AJAX, la mise à jour de vos Tweets sur votre téléphone, tout ça fonctionne sur le même principe.

Donc, sur votre site Web, Firefox, Chrome et les autres vont faire une requête à une machine, votre machine contient un code (si vous êtes dev, votre code :) qui va recupérer cette requête et générer une réponse.

Tout est texte

Généralement, les développeurs utilisent des outils sophistiqués pour écrire des sites Web. Si bien que quand on écrit du code, on ne voit pas vraiment la requête et la réponse, mais des fonctions, des objets, du HTML… Comment ça arrive et comment ça repart est géré automatiquement.

Regardons ce qui se passe quand ce n'est PAS géré automatiquement.

Voici le code d'un petit serveur HTTP écrit en Python 3. Il accepte n'importe quelle requête sur le port 7777 et retourne toujours une page avec marqué "Coucou".

```
import asyncio

# Le texte de la réponse qu'on va renvoyer
COUCOU = b"""HTTP/1.1 200 OK
Date: Fri, 16 Jun 2014 23:59:59 UTC
Content-Type: text/html

<html>
<body>
<h1>Coucou</h1>
</body>
</html>"""

# La fonction qui gère chaque requête et qui renvoie une réponse
# pour chacune d'entre elles. La même réponse à chaque fois pour cet
# exemple, mais on peut fabriquer une réponse différente si on veut.
```

- Beaucoup de femmes sont de mauvais coups (96)
- Une journée dans un FKK (maison close Allemande) (90)
- Fêtons le départ de Max (87)
- On est en 2013, on a marché sur la lune, mais on a toujours pas… (85)

TagCloud

bash bitcoin blog bottle cache comprehension-lists css cul decorator **django** email encoding error fun **git** html http import ipython iterable **javascript** jquery lambda linux **meta** mysql nginx nsfw php pip poo **python** redis ruby server shell ssh sublime text twitter **ubuntu** unicode unit tests unpacking virtualenv yield

All Tags

Archives

- June 2014 (14)
- May 2014 (14)
- April 2014 (10)
- March 2014 (23)
- February 2014 (27)
- January 2014 (17)
- December 2013 (27)
- November 2013 (25)
- October 2013 (27)
- September 2013 (1)
- August 2013 (31)
- July 2013 (32)
- June 2013 (24)
- May 2013 (30)
- April 2013 (31)
- March 2013 (33)
- February 2013 (29)
- January 2013 (34)
- December 2012 (32)
- November 2012 (36)
- October 2012 (35)
- September 2012 (36)
- August 2012 (37)
- July 2012 (33)

```
def handle_request(request, response):
    # On lit la requête
    data = yield from request.read(1000000)
    # On affiche son contenu. Surprise, c'est que du texte !
    print(data.decode('ascii'))
    # On écrit notre réponse, que du texte aussi !
    response.write(COUCOU)
    # On ferme la connexion : le protocole HTTP est stateless,
    # c'est à dire qu'il n'y a pas de maintien d'un état
    # côté client ou serveur et chaque requête est indépendante
    # de toutes les autres.
    response.close()

if __name__ == '__main__':
    # Machinerie pour faire tourner le serveur :
    # Récupération de la boucle d'événements.
    loop = asyncio.get_event_loop()
    # Création du serveur qui écoute sur le port 7777
    # et qui va appeler notre fonction quand il reçoit
    # une requête.
    f = asyncio.start_server(handle_request, port=7777)
    # Installation du serveur dans la boucle d'événements.
    loop.run_until_complete(f)
    # On démarre la boucle d'événement.
    print("Serving on localhost:7777")
    loop.run_forever()

# Si vous êtes dev, vous commencez à comprendre combien
# les libs et frameworks qui gèrent tout ce bordel pour
# vous sont fantastiques.
```

Si on lance le serveur et qu'on visite la page `http://localhost:7777` sur un navigateur, on va alors voir ceci dans le terminal où tourne le serveur :

```
$ python3 server.py
Serving on localhost:7777
GET / HTTP/1.1
Host: 127.0.0.1:7777
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:30.0) Gecko/20100101 Fi
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Cache-Control: max-age=0
```

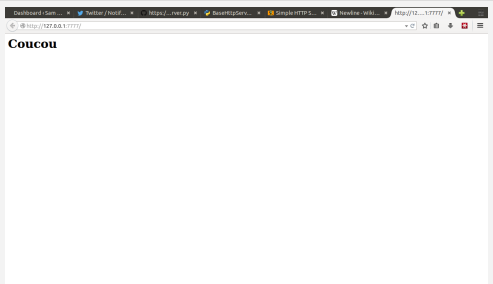
C'est la requête envoyée par notre client, et reçue par notre serveur. "Aller" sur l'adresse veut dire en fait que votre navigateur envoie ce morceau de texte.

Juste après, mon serveur renvoie aussi du texte. Toujours le même dans notre cas :

```
HTTP/1.1 200 OK
Date: Fri, 16 Jun 2014 23:59:59 UTC
Content-Type: text/html

<html>
<body>
<h1>Coucou</h1>
</body>
</html>
```

Le navigateur l'interprète, et vous fabrique cette page :



Vous contemplez le Web de 1995

Comme vous pouvez le voir, ce n'est que du texte tout simple qui est reçu et envoyé.

Quand quelqu'un développe un site Web, ce qu'il fait vraiment, c'est ça. La plupart du temps il ne le voit pas car ses outils lui facilitent la tâche en analysant le texte et en lui donnant des fonctions et des objets pour le manipuler. Mais derrière, c'est juste du texte.

June 2012 (24)
May 2012 (31)
April 2012 (20)
March 2012 (10)
February 2012 (14)

Flux RSS

RSS - Posts

Planet Python fr

Sélection d'articles sur Python du Web français

Envoyez des sioux

On adooooore les bitcoins :

19zAHPPuce4BAhsdy9KaFwVLurEJXMhMAn

Quand quelqu'un surf le Web, ce qu'il fait vraiment, c'est ça. Mais le navigateur se charge de le cacher derrière des jolis contrôles, et affiche un résultat bien plus agréable à regarder.

Structure des requêtes

Le texte d'une requête est divisé en 3 parties :

- L'action demandée sur la ressource.
- Les headers.
- Le corps de la requête

Prenons notre précédente requête d'exemple :

```
GET / HTTP/1.1
Host: 127.0.0.1:7777
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:30.0) Gecko/20100101 Fi
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Cache-Control: max-age=0
```

L'action demandée sur la ressource est toujours sur la première ligne :

```
GET / HTTP/1.1
```

Elle se divise en 3 parties :

- Le verbe d'action : GET, POST, PUT, OPTION, HEAD, etc. C'est ce qu'on veut faire sur les données qu'on demande. Le plus courant sur le Web, c'est GET (lire la donnée) et POST (créer la donnée, qu'on utilise souvent pour les formulaires).
- Le chemin de la ressource. Où se trouve la ressource à laquelle on souhaite accéder. Ici c'est "/", la racine, mais ça peut être "/user/monique/profile" par exemple.
- La version du protocole utilisé. Aujourd'hui tout le monde utilise HTTP en version 1.1, donc ça ne change pas vraiment.

Notez bien que le chemin de la ressource n'a pas à correspondre à un chemin réel d'un fichier sur l'ordinateur. Une ressource est quelque chose de complètement virtuel, quelque chose que mon programme met à disposition selon mes désirs. Si c'est un fichier, très bien, mais ce n'est pas obligatoire, et je peux générer n'importe quelle réponse qui me plait.

En effet, si je vais sur l'adresse `http://127.0.0.1:7777/user/monique/profile/`, vous notez que mon serveur continue de marcher. Il reçoit simplement la requête :

```
GET /user/monique/profile/ HTTP/1.1
...
```

C'est à mon serveur de décider ce qu'il choisit de faire avec le chemin `/user/monique/profile/`. Ici il est un peu con et continue de renvoyer "coucou".

Bien, on vient de voir "L'action demandée sur la ressource", voyons maintenant les headers :

```
Host: 127.0.0.1:7777
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:30.0) Gecko/20100101 Fi
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Cache-Control: max-age=0
```

Les headers sont des informations supplémentaires que le client fourni à mon serveur sous la forme :

```
Nom-De-L-Information: contenu
```

Les sauts de ligne sont très importants. La première ligne de la requête est "L'action

demandée sur la ressource", puis on met un saut de ligne, ensuite vient un header, puis un saut de ligne, puis un header… Chaque header doit tenir sur une ligne.

En HTTP/1.1, seul le header `Host` est obligatoire, mais comme aucun header n'était obligatoire en HTTP/1.0, beaucoup de serveurs acceptent l'absence de tout header.

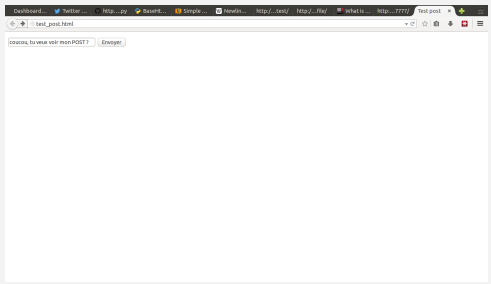
Les headers contiennent généralement des informations sur le client (ex: `Accept-Language` vous dit quelles langues le client accepte), le contenu de la requête (ex: `Content-Length` indique la taille de la requête) ou demande un comportement du server (`Cache-Control` précise comment gérer les ressources qu'on peut mettre en cache).

Pour "Le corps de la requête", il nous faut ajouter du contenu à notre requête.

Pour cela, faisons une page avec un petit formulaire HTML :

```
<html>
<head>
    <title>Test post</title>
</head>
<body>
<form method="post" action="http://127.0.0.1:7777/">
<p><input type="test" value="coucou, tu veux voir mon POST ?" name="salut">
<input type="submit"></p>
</form>
</body>
</html>
```

Ce qui nous donne cette page :



Techniquement, on peut poster un formulaire avec une requête GET, mais shut...

Si on active le formulaire, notre serveur affiche cette requête :

```
POST / HTTP/1.1
Host: 127.0.0.1:7777
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:30.0) Gecko/20100101 Fi
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 41

salut=coucou%2C+tu+veux+voir+mon+POST+%3F
```

Déjà, vous notez que le verbe d'action a changé : `POST / HTTP/1.1`.

Ensuite, à la fin de la requête, on laisse une ligne vide, puis on a de nouveau du texte :

```
salut=coucou%2C+tu+veux+voir+mon+POST+%3F
```

C'est le corps de la requête.

C'est comme ça que le serveur reçoit le contenu des données des formulaires, et c'est ce qu'on retrouve dans la variable `$_POST` en PHP ou `request.POST` en Django

Encore une fois, les outils de programmation évitent au codeur de travailler avec du charabia, et le transforme en quelque chose de facile à comprendre et à manipuler.

Structure des réponses

C'est pareil, ma bonne dame. Reprenons notre exemple de réponse :

```
HTTP/1.1 200 OK
Date: Fri, 16 Jun 2014 23:59:59 UTC
Content-Type: text/html

<html>
<body>
<h1>Coucou</h1>
</body>
</html>
```

Première ligne, on précise le protocole, puis le **code de réponse**, qui stipule la nature de votre réponse (tout va bien, une erreur, une redirection, la page n'existe pas, etc).

Ensuite les headers, puis on saute une ligne, et on met le corps de la réponse. Ici, le code HTML qui va donner notre jolie page "Coucou".

Tout tient là dedans

Tout le reste, toutes les fonctionnalités fantastiques du Web (les liens hypertextes, les fichiers statiques JS et CSS inclus, les cookies, les redirections, le cache, la compression…) ont pour point d'entrée un de ces 3 éléments de la requête ou de la réponse. C'est que c'est un protocole bien foutu.

Enfin, disons, **presque tout** ? :)

Télécharger le code de l'article

Partager: Email More

No related posts.

Flattr this!

6 comments

FLOZz
18/06/2014

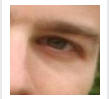
Il y a une petite erreur dans le premier schéma : c'est « 401 Unauthorized », pas 301 :)

Sam
18/06/2014

Et merde, 301 c'est la redirection (en plus je viens juste de faire un article dessus). J'ai trop la flemme d'éditer le dessin, le reuploader, changer les deux articles… Tant pis, ça restera là.

Romain
18/06/2014

Merci pour l'exemple d'utilisation d'asyncio ;)



foX
18/06/2014

> *# On ferme la connexion : le protocole HTTP est stateless,*
> *# c'est à dire qu'il n'y a pas de maintien d'un état*
> *# côté client ou serveur et chaque requête est indépendante*
> *# de toutes les autres.*

Presque ;-). Je chipotte, mais justement en HTTP 1.1 c'est plus sioux. Au niveau HTTP en effet, les requêtes sont indépendantes. La session est gérée (ou pas !) au niveau de l'application par

divers moyens, et les fameux cookies sont un des moyens permettant de garder l'état pour savoir qui est qui et recoller les bouts.
Mais au niveau TCP, ce n'est heureusement pas stateless. Donc après la réponse, le serveur ne fait pas un close() sur la socket TCP, et le client non plus. Cela permet d'envoyer plusieurs requetes HTTP dans la même connexion TCP.

Pourquoi ça ? Et bien pour des raisons de perf. L'ouverture d'une connexion TCP a un coût (syn, ack, synack) et il est plus rapide de garder les tuyaux ouverts entre un client et le serveur web, une page page web étant souvent constituée de dizaines de ressources demandant chacune un appel HTTP. De plus, cela limite le nombre de connexion TCP en //, ce qui a un coût au niveau de tous les équipements réseaux sur le chemin de votre requête (firewal, reverse proxy, …).

C'est pour cela que le header sur la taille de la réponse est obligatoire en HTTP 1.1, car le client doit savoir quand ça s'arrette. En HTTP 1.0, c'est facile, il lit jusqu'à qu'à recevoir un close() de la part du serveur. Mais en HTTP 1.1 la socket reste ouverte, donc le client doit lire jusqu'à x octets, ensuite la connexion reste ouverte pour envoyer une autre requête.

On peut même multiplexer plusieurs requêtes dans une même connexion TCP, c'est à dire que le client fait plusieurs demandes à la suite sans attendre la réponse de chacune, puis reçoit plusieurs réponses, au lieu de simplement sérialiser. C'est plus rock'n roll et pas toujours pris en charge (souvent désactivé dans les browser). Cela s'appelle le pipelining http.

policier moustachu
18/06/2014

the schumacher song !!! Depuis le temps que je recherchais ce titre. Merci les gars.

policier moustachu
18/06/2014

J'adore la techno kitsh. Autre titre phare : le theme de Mortal Kombat

Flux RSS des commentaires

Leave a Reply

Your email address will not be published. Required fields are marked *

Name *

Email *

Website

Comment

You may use these HTML tags and attributes: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code> <del datetime=""> <em> <i> <q cite=""> <strike> <strong> <pre lang="" line="" escaped="" highlight="">`

Submit Comment

☐ Notify me of follow-up comments by email.

☐ Notify me of new posts by email.

Jouer à mario en attendant que les autres répondent

contactez-nous

Designed by Elegant WordPress Themes